Développement Mobile Android Audio Player

Anthony Chomienne

CPE Lyon 2021

Audio Player

Cette application a pour but de permettre à des utilisateurs d'écouter la musique présente sur leur téléphone et de pouvoir récupérer les informations sur un artiste ou un album depuis un webservice où même des playlists déjà existantes sur un service en ligne.

Objectifs

- Prendre en main Android
- Développer et Déployer une application native (java)
- Voir les différentes étapes du développement Android
- Utiliser un webservice REST
- Connaître un peu mieux le coeur d'Android

Note:

Cette application réalisée sur les 9 séances de TP aura pour but de vous faire voir un certain nombre d'aspect de la programmation pour Android en Java. On utilisera pas de bibliothèque externe afin de vous montrer ce qui se passe au cœur de votre système. L'objectif est de vous montrez comment faire vous-même des opérations qui n'ont pas de bibliothèque adaptée ou dont vous ne pouvez pas vous servir pour que vous ne soyez pas complètement démunis. Les séances ne sont pas découpées dans le sujet. Plus vous avancez plus vous pourrez ajouter des fonctionnalités bonus. L'évaluation tiendra compte de l'avancement, de la rigueur que vous avez appliquée et de la qualité du travail fourni.

Premier Pas

Lors de la création d'uni nouveau projet sous Android Studio. Il vous faudra un nom et un company name. Ces deux informations serviront à définir le package de base de votre application, c'est également l'un des moyens qu'ont

les smartphones d'identifier les différentes applications. Il doit donc être unique. Pour le choix de la version minimum d'Android, vous avez le choix. Android Studio vous donne une approximation du nombre de terminaux actifs sur le playstore supporté à partir de cette version. Pour avoir une version à jour: https://developer.android.com/about/dashboards/index.html.

Actuellement selon les besoins, je choisis une version minimale 19 correspondant à KitKat. La quantité de personnes ne pouvant pas utiliser l'application étant de toute façon minimal.

Choisissez ensuite comme activity de départ: Empty Activity. Ceci vous donnera déjà une base de code pour commencer.

L'arborescence de votre projet sera alors la suivante:

- app
  - manifests
  - java
  - res
- Gradle Scripts
  - build.gradle (Project: JourneyDiaries)
  - build.gradle (Module: app)

La partie **manifests** contient le fichier AndroidManifest.xml. La partie **java** contient les packages de votre projet et vos sources java. Par défaut vous n'avez qu'un package *company_name.app_name*. La partie **res** contient les différentes ressources de votre projet réparties dans des dossiers appropriés.

Il y a un dossier **drawable**, **layout**, **mipmap** et **values** au sein du dossier res. Ces dossiers peuvent être suffixés par divers élément comme vue en cours.

drawable contient les images de votre projet. **layout** contient les fichiers xml permettant de décrire les Vues au sens MVP du terme. **mipmap** contient l'icones de l'application dans différentes résolutions. **values** contient les fichiers de style, les fichiers de textes...

Lorsque vous créez le projet par défaut, Android Studio vous propose de créer une activity plus ou moins vide en fonction de ce que vous avez choisi. Dans notre cas, si on regarde le fichier *res/layout/activity_main.xml* (ou autre si vous lui avez donné un autre nom), elle contient déjà beaucoup de choses : Un ConstraintLayout et une TextView contenant «Hello World ! »

Les deux fichiers build.gradle servent à l'outil de compilation Gradle. Le second contient en particulier différentes informations concernant votre application (voir cours).

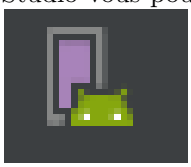
Build

Le build se fait automatiquement lorsque vous lancez votre application, mais vous pouvez le lancer manuellement depuis le menu Build. Pour déployer sur un

téléphone, vous avez deux cas:

- Utilisez un Android Virtual Device (AVD).
- Utilisez un device Android.

Android Virtual Device (AVD)

Dans la barre d'outils d'Android Studio vous pouvez voir à droite un certain nombre d'îcones dont la suivante: 

Une fenêtre s'ouvre contenant vos devices virtuels présents sur votre système (Normalement aucun) et vous offre la possibilité d'en créer d'autres en partant d'une base existante (Nexus 5, Nexus 4, Pixel...), vous pouvez également choisir la version de votre device afin de tester sur différentes versions du système.

Android Device

Deux éléments sont nécessaires pour développer en utilisant son téléphone:

- Autoriser l'installation depuis des sources inconnus

Rendez-vous dans les paramètres du téléphone, dans le menu sécurité et activer l'autorisation d'installer des applications depuis une source inconnue.

- Activer le mode développeur et notamment le debuggage USB

Pour activer le mode développeur, dans les paramètres du téléphone, allez dans «À propos du téléphone» puis «Information sur le logiciel» et appuyer plusieurs fois sur le «numéro de build», au bout d'un moment il va vous dire qu'il a débloqué les options développeur. Vous pourrez alors vous rendre dans les options développeur pour activer le débuggage USB.

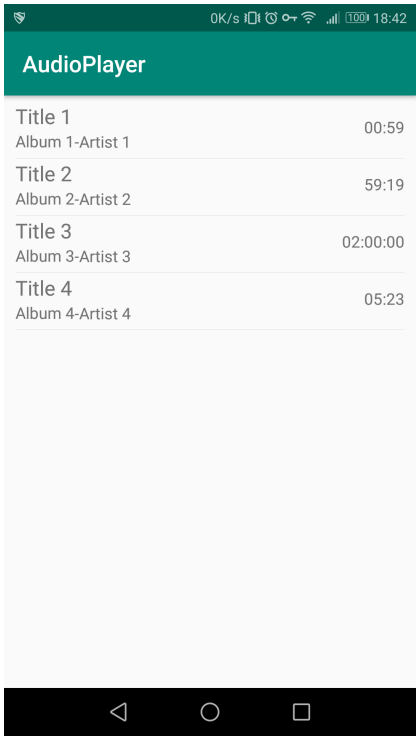
Run

Un clic sur Run dans l'IDE ou sur Debug installe l'application sur le device que vous séléctionnerez dans la fenêtre qui s'ouvrira alors. Après l'installation, il lancera votre application sur le téléphone. Il existe depuis quelque temps, l'**Instant Run**. Il envoie les modifications et ne relance pas l'application depuis zéro. Ceci a, je trouve, tendance à mettre les applications dans des états pas toujours correct (variable pas dans le bon état...). À vous de voir si vous voulez l'utiliser ou non. (Sur les machines de CPE vous ne pourrez pas l'utiliser, désactivez-le donc dans les paramètres d'Android Studio)

Première Activity, Liste et DataBinding

Nous allons commencer cette application par une liste de fichier audio (AudioFile) visible sur l'écran de démarrage de l'application. Cet écran permettra également d'accéder à un album, artiste à partir d'un titre. Dans un premier temps, nous nous contenterons d'afficher une liste statique ne représentant pas encore des fichiers sur le disque. Nous utiliseront le databinding et le ViewModel pour ça.

L'ojectif est d'obtenir quelque chose comme suit :



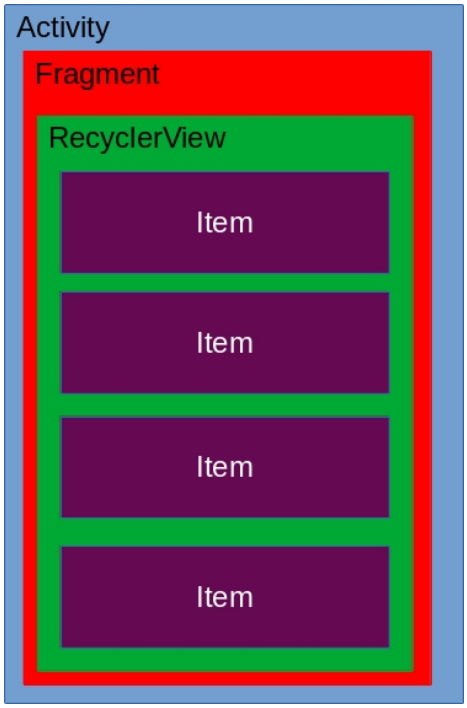
Pour commencer dans le fichier *build.gradle* du dossier *app*. Ajouter dans la balise *android*, la balise **buildFeatures** avec **dataBinding true**, comme suit:

```
android {
  ...

  buildTypes {
    ...
  }
  buildFeatures {
      dataBinding  true
  }
}
```

Synchronisez le projet après avoir fait cette modification

Le schéma suivant vous présente les différentes strate de l'interface que nous allons mettre en place.



N'hésitez pas à à reproduire ce schéma et à le compléter d'annotation pour vous aider à comprendre l'architecture des différents éléments par rapport a votre code.

Nous allons maintenant modifier le fichier **activity_main.xml** afin de mettre en place le databinding pour celui-ci ainsi que de préparer l'utilisation de fragment.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    >
    <data/>
    <LinearLayout
        android:background="#0000FF"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
    <FrameLayout
        android:layout_margin="5dp"
        android:id="@+id/fragment_container"
        android:layout_width="match_parent"
```

```
            android:layout_height="match_parent"
            />
        </LinearLayout>
    </layout>
```

Le FrameLayout ici nous servira de point d'ancrage pour nos différents Fragments. Il est ici, pour être remplacé par les fragments successifs que nous utiliserons à l'aide de transactions.

MainActivity.java du fait de l'utilisation du DataBinding, va devoir évoluer et ressembler à ce qui suit.

```
public class MainActivity extends AppCompatActivity {

private ActivityMainBinding binding;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    binding = DataBindingUtil.setContentView(this, R.layout.activity_main);
}
```

Nous voulons manipuler des fichiers musicaux. Un fichier musical correspond à un titre, un chemin, une durée, un artiste, un album, une année, un genre... Nous aurons donc comme classe modèle la classe AudioFile qui suit.

```
public class AudioFile {
  private String title;
  private String filePath;
  private String artist;
  private String album;
  private String genre;
  private int year;
  private int duration;

  public String getTitle() {
      return title;
  }

  public void setTitle(String title) {
      this.title = title;
  }

  public String getFilePath() {
      return filePath;
  }

  public void setFilePath(String filePath) {
```

```
    this.filePath = filePath;
}

public String getArtist() {
    return artist;
}

public void setArtist(String artist) {
    this.artist = artist;
}

public String getAlbum() {
    return album;
}

public void setAlbum(String album) {
    this.album = album;
}

public String getGenre() {
    return genre;
}

public void setGenre(String genre) {
    this.genre = genre;
}

public int getYear() {
    return year;
}

public void setYear(int year) {
    this.year = year;
}

public int getDuration() {
    return duration;
}

public String getDurationText() {
    int second = duration % 60;
    int durationMinute = (duration - second) / 60;
    int minute = durationMinute % 60;
    int hour = (durationMinute - minute) / 60;

    if(hour > 0)
```

```
            return String.format(Locale.getDefault(),
        "%02d:%02d:%02d",hour,minute,second);
        return String.format(Locale.getDefault(),
        "%02d:%02d",minute,second);
    }

    public void setDuration(int duration) {
        this.duration = duration;
    }
    }
```

Créez un second layout nommé **audio_file_list_fragment.xml** il contiendra ce qui suit:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">
    <data/>
    <androidx.constraintlayout.widget.ConstraintLayout
        android:background="#FF0000"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        >

        <androidx.recyclerview.widget.RecyclerView
            android:background="#00FF00"
            android:id="@+id/audio_file_list"
            android:layout_width="0dp"
            android:layout_height="0dp"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            android:layout_marginStart="10dp"
            android:layout_marginEnd="10dp"
            android:layout_marginTop="5dp"
            android:layout_marginBottom="5dp"
            />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Ce layout représente l'écran d'accueil de l'application. Une liste des morceaux présent sur le téléphone. La RecyclerView ajouté il y a quelques années est mieux que la ListView en termme de gestion de la mémoire mais son fonctionnement est similaire. Il automatise certaine chose qu'il fallait faire manuellement avant.

De la même façon que pour l'Activity, nous avons besoin d'un fichier Java

associé. Celui-ci étendra le type Fragment. Créez le fichier **AudioFileListFragment.java**. Il contiendra dans un premier temps ce qui suit:

```
public class AudioFileListFragment extends Fragment {
    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater,
                        @Nullable ViewGroup container,
                        @Nullable Bundle savedInstanceState) {

        AudioFileListFragmentBinding binding = DataBindingUtil.inflate(inflater,
                        R.layout.audio_file_list_fragment,container,false);
        binding.audioFileList.setLayoutManager(new LinearLayoutManager(
                        binding.getRoot().getContext()));
        return binding.getRoot();
    }
}
```

La fonction getRoot de AudioFileBinding nous renvoie la view Java après avoir été créé à partir du fichier xml par la fonction inflate.

Pour afficher un fragment dans votre activity comme vu en cours. A partir de là vous pouvez lancer et voir certain résultat sur votre téléphone ou émulateur. Pensez à appeler cette fonction!

```
public void showStartup() {
      FragmentManager manager = getSupportFragmentManager();
      FragmentTransaction transaction = manager.beginTransaction();
      AudioFileListFragment fragment = new AudioFileListFragment();
      transaction.replace(R.id.fragment_container,fragment);
      transaction.commit();
}
```

Que vous utilisiez une ListView ou une RecyclerView, il faudra un Adapter. Un Adapter est le morceau de code qui va servir à indiquer à la liste les éléments présents dans celle-ci, leur nombre mais aussi la façon dont ils seront affichés. Il éxiste des Adapter basiques, mais nous voulons un affichage un peu plus compliqué et personnalisé que ce que permette les adapter de base. L'adapter s'occupe en fonction de l'affichage de créer le bon nombre d'élément affichable et de les recycler si ceux-ci ne sont plus visibles. Si notre liste contient 50 titres et que la vue ne peut en afficher que 10 à la fois. L'Adapter créera seulement 11 ou 12 vue élément. Quand une vue sera hors écran suite à un défilement il la réutilisera pour le nouvel élément qui apparaît.

Pour gérer la disposition des éléments d'un titre à afficher, nous allons devoir définir un nouveau layout **audio_file_item.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    xmlns:app="http://schemas.android.com/apk/res-auto"
>
<data/>
<androidx.constraintlayout.widget.ConstraintLayout
    android:background="#FF00FF"
    android:layout_margin="5dp"
    android:layout_width="match_parent"
    android:layout_height="50dp">

    <TextView
        android:textAppearance="@style/TextAppearance.AppCompat.Medium"
        android:id="@+id/title"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toStartOf="@id/duration"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toTopOf="@id/album"
        android:layout_width="0dp"
        android:layout_height="wrap_content"/>
    <TextView
        android:id="@+id/album"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toStartOf="@id/dummy"
        app:layout_constraintBottom_toTopOf="@id/separator"
        app:layout_constraintTop_toBottomOf="@id/title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <TextView
        android:id="@+id/dummy"
        android:text="-"
        app:layout_constraintStart_toEndOf="@id/album"
        app:layout_constraintEnd_toStartOf="@id/artist"
        app:layout_constraintBottom_toTopOf="@id/separator"
        app:layout_constraintTop_toBottomOf="@id/title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

    <TextView
        android:id="@+id/artist"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toTopOf="@id/separator"
        app:layout_constraintEnd_toStartOf="@id/duration"
        app:layout_constraintStart_toEndOf="@id/dummy"
        app:layout_constraintTop_toBottomOf="@id/title" />
    <TextView
        android:id="@+id/duration"
```

```
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            app:layout_constraintBottom_toTopOf="@id/separator"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <TextView
            android:id="@+id/separator"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintBottom_toBottomOf="parent"
            style="?android:attr/listSeparatorTextViewStyle"
            android:layout_width="match_parent"
            android:layout_height="5dp"/>
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Passons à l'adapter. Ce dernier va recevoir en entrer la liste des données à afficher. La fonction onCreateViewHolder sera appelé pour chaque vue à créer (nombre limité évoqué plus haut). On ne passera que peu de fois dans la fonction. A contratio la fonction onBindViewHolder est quant à elle appelé pour remplir la vue associée au ViewHolder avec les données d'un élément. Si on fait défiler la vue, les ViewHolder plus visible à l'écran seront recyclé pour prendre la place de ceux s'affichant en dessous des précédents. On reçoit ici la position de l'élément à afficher. La fonction getItemCount permet de donner l'information à la recyclerView sur le nombre maximum d'élément et du coup savoir quand on est à la fin de la liste ou non.

```
public class AudioFileListAdapter extends
          RecyclerView.Adapter<AudioFileListAdapter.ViewHolder> {

    List<AudioFile> audioFileList;

    public AudioFileListAdapter(List<AudioFile> fileList) {
        assert fileList != null;
        audioFileList = fileList;
    }

    @NonNull
    @Override
    public ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        AudioFileItemBinding binding = DataBindingUtil.inflate(
                    LayoutInflater.from(parent.getContext()),
                    R.layout.audio_file_item, parent,false);
        return new ViewHolder(binding);
    }

    @Override
```

```
  public void onBindViewHolder(@NonNull ViewHolder holder, int position) {
      AudioFile file = audioFileList.get(position);
      holder.binding.title.setText(file.getTitle());
      holder.binding.artist.setText(file.getArtist());
      holder.binding.album.setText(file.getAlbum());
      holder.binding.duration.setText(file.getDurationText());
  }

  @Override
  public int getItemCount() {
      return audioFileList.size();
  }

  static class ViewHolder extends RecyclerView.ViewHolder {

      private AudioFileItemBinding binding;

      ViewHolder(AudioFileItemBinding binding) {
          super(binding.getRoot());
          this.binding = binding;
      }
  }
}
```

Pour affecter notre adapter à notre liste nous aurons besoin d'ajouter dans le fragment les lignes suivantes. Remplacez-les ... par la création d'élément servant à alimenter la liste avec des données statiques. Nous verrons plus tard pour récupérer des données réelles.

```
List<AudioFile> fakeList = new ArrayList<>();
...
binding.audioFileList.setAdapter(new AudioFileListAdapter(fakeList));
```

À faire:

- Mettez en place cette liste, n'hésitez pas à personnaliser l'aspect et/ou la disposition des éléments pour correspondre à votre envie/besoin
- Esayer de bien comprendre tout le code fourni et comment fonctionnent les intéractions entre les différents éléments, comment ils sont liés entre eux. Vous aurez besoin de réappliquer ces éléments pour d'autres parties de l'application
- Construisez un jeu d'essai qui vous servira à tester l'affichage des éléments et si votre liste s'affiche correctement.

Model - View - ViewModel

Comme vu en cours, le modèle c'est AudioFile. Notre view c'est audio_file_item.xml. Il nous reste à définir notre View-Model. Pour avoir notre ensemble complet. Le ViewModel communique à la fois avec la view et à la fois avec le model.

Notre ViewModel est assez simple, nous n'avons que des champs à afficher dans un premier temps. Le ViewModel étend BaseObservable. Ceci afin de pouvoir raffraichir la vue si des données étaient amenés à changer. Ce sera notre cas. C'est l'annotation @Bindable qui indique qu'un champ peut être mis à jour. Il peut être mis à jour dans une fonction setXXX et il existe deux méthodes pour notifier ce changement notifyChange() qui indique que tout a été changé et qu'il faut donc mettre à jour tous les champs Bindable ou notifyPropertyChanged(BR.propertyName) pour indiquer qu'une propriété en particulier à changer. (BR => BindableResource, fichier généré comme le fichier R). Notre cas sera le premier, si on change de fichier dans le ViewModel, il faut obligatoirement que tous les champs s'y rapportant soit mis à jour.

```
public class AudioFileViewModel extends BaseObservable {
  private AudioFile audioFile = new AudioFile();

  public void setAudioFile(AudioFile file) {
      audioFile = file;
      notifyChange();
  }

  @Bindable
  public String getArtist() {
      return audioFile.getArtist();
  }

  @Bindable
  public String getTitle() {
      return audioFile.getTitle();
  }

  @Bindable
  public String getAlbum() {
      return audioFile.getAlbum();
  }

  @Bindable
  public String getDuration() {
      return audioFile.getDurationText();
  }
}
```

Pour créer le lien entre la vue et le viewmodel nous avons la balise data que nous avons laissé vide dans nos layout. Ce ne sera plus le cas pour **audio_file_item.xml**. audioFileViewModel est le nom de la variable que l'on va utiliser (vous pouvez l'appeler comme vous voulez) et son type est la classe AudiFileViewModel que nous avons défini au-dessus.

```
<data>
      <variable
      name="audioFileViewModel"
      type="fr.mobdev.audioplayer.AudioFileViewModel"/>
  </data>
```

Pour l'utiliser dans un layout, il n'y a rien de bien compliqué. Là où vous avez besoin de cette variable. Ci-dessous pour mettre le titre dans la textView vous écrivez @{nomVariable.donnee}. La fonction getDonnee() du ViewModel sera donc appelé pour remplir ce champ.

```
<TextView
    android:id="@+id/title"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="@{audioFileViewModel.title}"
    android:textAppearance="@style/TextAppearance.AppCompat.Medium"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toStartOf="@id/duration"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toTopOf="@id/album"/>
```

Dernière étape pour finir c'est donner au databinding le ViewModel afin qu'il puisse l'utiliser avec la vue. Dans la fonction onBindViewHolder nous avions un certain nombre de fonction setText(). Nous allons les remplacer par

```
holder.viewModel.setAudioFile(file);
```

Dans le viewHolder nous allons ajouter la création du ViewModel qui sera une variable membre et l'affecter au binding.

```
static class ViewHolder extends RecyclerView.ViewHolder {

      private AudioFileItemBinding binding;
      private AudioFileViewModel viewModel = new AudioFileViewModel();

      ViewHolder(AudioFileItemBinding binding) {
          super(binding.getRoot());
          this.binding = binding;
          this.binding.setAudioFileViewModel(viewModel);
      }
}
```

Communication Fragment vers Activity

Pour communiquer entre fragment ou entre un fragment et l'activity plusieurs méthodes sont possibles. La plus simple consite à créer un Listener qui servira d'outil d'échange de donnée entre deux composant. Le listener sera initialisé à l'endroit où l'on a besoin de faire remonter une information et sera transmis à l'endroit où est produit l'information afin de pouvoir être déclanché au bon moment.

Un listener c'est une interface définissant une ou plusieurs méthodes pouvant être déclenché afin de remonter une ou plusieurs informations. Dans l'exemple qui suit nous avons un listener avec une méthode onDoneSomething qui sera déclenché à la fin d'un traitement.

```
public interface MyListener{
  public void onDoneSomething();
}
```

A l'endroit où vous allez vouloir remonter l'information vous allez devoir implémenter le listener. Ce sera souvent dans une Activity ou dans un fragment (dépendant de votre besoin). Ici lorsque le listener sera déclenché, la fonction someCode sera éxécuté.

```
MyListener listener = new MyListener(){
  @Override
  public void onDoneSomething(){
    someCode();
  }
};
```

La transmission de ce listener au fragment qui va emmetre l'information.

```
fragment.setMyListener(listener);
```

Dans le fragment qui doit transmettre une information, nous aurons donc quelque chose qui ressemble à ce qui suit.

```
private MyListener myListener;
public void setMyListener(MyListener listener)
{
  myListener = listener;
}
```

Dans ce même fragment nous avons par exemple une fonction someFunction qui prend un certain temps et à la fin de l'exécution de celle-ci nous voulons remonter l'information que tout s'est bien déroullé à notre activity, pour cela nous utilisons donc le listener que nous avons reçu. À la fin de someFunction, on appelle onDoneSomething tel qu'il a été implémenté. C'est à dire qu'il appelera la méthode de l'activity qui s'appelle someCode.

```
public void someFunction() {
```

```
  doSomethingTakeSomeTime();
  if (myListener != null)
      myListener.onDoneSomething();
}
```

Si vous avez plusieurs couche à traverser. Activity <- Fragment <- Fragment rien ne vous empèche de transmettre le listener jusqu'à là où il est nécessaire.

L'avantage du listener c'est qu'il ne crée pas un couplage fort entre Activity et Fragment. On peut tout à fait réutiliser le fragment avec une autre activity sans que cette dernière ne s'intéresse a l'évènement produit à la fin de someFunction auquel cas elle ne crée pas de listener et ne le transmet pas au fragment. Le fragment n'a aucune connaissance du fait qu'il est dans une activity ou dans un fragment quelconque, il ne sait pas qu'il a des échanges avec quelqu'un d'autre. il émet une information éventuellement si on lui donne un canal de communication: le listener, mais il ne sais pas qui est au bout.

À faire

- Utiliser MVVM dans votre projet en suivant ce qui est indiqué au-dessus et continuer à l'utiliser par la suite pour les autres vues qui en auraient besoin
- Créer un second fragment qui s'affichera juste en dessous de la liste. Ce fragment représentera les différents boutons controllant la musique (play/pause, chanson suivante/précédente)

Accès aux fichiers et métadonnées

Pour accéder aux fichiers présents sur le téléphone vous aurez besoin de demander la permission READ_EXTERNAL_STORAGE à l'utilisateur. Ceci se fera dans MainActivity. Sans cette permission, nous ne pouvons pas faire grand-chose. Une fois la permission obtenue il faudra obtenir les fichiers audios, pour cela nous allons passer par le ContentResolver. Il lui faut une uri (chemin à partir duquel il cherche) et ce que nous voulons comme donnée sous forme de tableau.

```
Uri uri = MediaStore.Audio.Media.EXTERNAL_CONTENT_URI; // La carte SD
String[] projection = {MediaStore.Audio.Media.DATA, MediaStore.Audio.Media._ID,
                       MediaStore.Audio.Media.TITLE, MediaStore.Audio.Media.ARTIST};
                       //chemin du fichier, ID, titre, artist
Cursor cursor = context.getContentResolver().query(uri,projection,null,null,null)
...
//path peut être utilisé directement avec Mediaplayer.setDataSource(path)
String path = cursor.getString(0);
//audioFileUri doit être utilisé avec
//MediaPlayer.setDataSource(context,audioFileUri);
//Pour Android 10
Uri audioFileUri = ContentUris.withAppendedId(
```

```
                    MediaStore.Audio.Media.EXTERNAL_CONTENT_URI,
                    cursor.getLong(1));
...
```

Dans mon cas voici, ce que j'obtiens après traitement du curseur.

À faire

- Demandez la permission d'accéder aux stockages externes
- Mettez en place la recherche des fichiers sur le téléphone (Interne/externe) en choisissant bien où placer le code exemple et complétez-le.
- Vous aurez besoin de comprendre l'utilisation de la classe Cursor qui permet de parcourir le résultat de la recherche

Fonction Audio

Pour cette partie, vous aurez à utiliser le MediaPlayer. Pour cette partie vous êtes invité à regarder en détail https://developer.android.com/guide/topics/media/mediaplayer.html. La partie concernant les services est importante si vous voulez que votre player fonctionne écran éteint ou en arrière plan. Vous aurez à vous réferer aux cours et à https://developer.android.com/guide/components/bound-services pour la gestion du service.

Je vous invite à avoir la structure suivante pour la gestion du service, si vous voulez passer par là. Ce n'est pas obligatoire et pas forcément évident.

```
public class PlayerService extends Service implements MediaPlayer.OnErrorListener, MediaPlay

  private final Binder binder = new PlayerBinder();

  public int onStartCommand(Intent intent, int flags, int startId) {
      return START_STICKY_COMPATIBILITY;
  }

  public void play(String path) throws IOException {
  }

  public void stop() {
  }

  @Nullable
  @Override
  public IBinder onBind(Intent intent) {
      return binder;
  }

  public void onPrepared(MediaPlayer player) {
  }

  @Override
  public boolean onError(MediaPlayer mp, int what, int extra) {
      return false;
  }

  public class PlayerBinder extends Binder {
      public PlayerService getService() {
          return PlayerService.this;
      }
  }
}
```

À faire

- Permettre de jouer de la musique lorsque l'on clique sur un morceau
- Jouer la musique suivante quand la précédente est terminée
- Mettre à jour les contrôles en fonction des états de lecture

Webservice

Pour améliorer l'expérience utilisateur et proposer de nouveau contenu nous allons utiliser l'api de last fm: https://www.last.fm/api/. Pour cela vous devrez créer un compte et vou enregistrer ici: https://www.last.fm/api/account/create afin d'obtenir la clef d'API vous permettant de faire des requêtes auprès du webservice.

Vous retrouver la documentation de l'api ici https://www.last.fm/api/

À faire

- Mettez en place une récupération d'information à l'aide de l'API pour enrichir le contenu de votre application

Base de données

Vous avez récupéré des données depuis le webservice, maintenant il faut les enregistrer en base de données. Établissez le schéma de votre base de données afin de stocker ces données.

À faire

- Établissez le schéma de la base de donnée pour votre application
- Enregistrez les informations dans votre base de données
- Affichez les données depuis votre base de donnée
- Pensez à protéger l'accès à la base de donnée pour ne pas la rendre accessible à toute l'application. Uniquement quelques fonctions doivent être accessibles.

Nouvelles pages et Navigation

À faire

- Ajoutez différentes pages afin d'afficher les différentes données récupérer depuis le webservice et d'améliorer l'expérience utilisateur.
  - Liste des différents artistes
  - Liste des différents genres
  - liste des différents albums

Ces pages pourront être accessibles de différentes manières. Libre à vous de choisir une méthode adaptée.

Fonctionnalités en plus

- Gestion de playlist

- Fonction de recherche
- Gérer une notification permanente reprenant les contrôles audio
- Suggestion personnalisé à partir des artistes similaires proposé par l'API et des artistes présents sur le téléphone.

Intégration Continue

Pour que l'intégration continue dans votre projet il faut créer dans votre dépot le fichier .drone.yml avec le contenu suivant:

```
---
kind: pipeline
type: exec
name: default

steps:
- name: build
  commands:
  - bash ./gradlew assembleDebug
```